Assignment 2: Board Game (X vs Y) 2018 Semester 1

30 MARKS

The aim of this assignment is for students to:

- **Analyse and document requirements and produce suitable design documents**
- **Develop an awareness of good object oriented design principles (GoF Design Patterns)**
- **Design and write maintainable code through an iterative process and refactoring**
- **Use a disciplined approach to team development through design by contract**
- **Work successfully in a team**

Part II (Specification) (30 marks)

Assignment Part II is designed to exercise the Gamma et al. design pattern taught from weeks 5 to 10. For this assignment you are also required to provide additional game features that make the game more functional and playable. You should still follow all of the code guidelines provided in Appendix A of the first assignment specification.

A. Presentation and Design Documents (6 marks)

(to be presented during week 12)

You are required to present your designs and justify your design decisions *(emphasis on the use of specific GoF design patterns)* as well as demonstrate a functioning game. You may also reflect on any other lessons that you learnt. You may use ppt slides for your presentation as well as a laptop showing your implementation (10 mins including question time).

B. Additional Functional Requirements (12 marks)

Complete the implementation for all of the requirements in Part I and implement the following **two mandatory** extra functionalities:

- Incorporate an undo moves option. Each player can undo from 1 to 3 moves in one go but can exercise this option only once per game. Undoing affects both players regardless of who initiated it. For example if Player 1 undoes the last two moves then both players last two moves are cancelled and Player 1 now plays their turn.
- Add an additional combat capability to each piece (players select the mode e.g. fighting stance versus defensive stance) before each move.

In addition, **two** of the following extra functionalities or graphical user interface requirements must be met to get full marks.

- An option to save the game state midway and restart later.
- An option to create different sized boards and varying number of pieces.
- An option to introduce obstacles onto the game board which limits available moves.
- An option to replay all the moves of the current game visually and slowed down to the speed of human comprehension.

- Drag and drop features for moving a piece.
- Allow playing against the system. Since this is not an AI course you can keep the "AI" logic simple i.e. it does not matter if the system does not always win!
- Extend the game to a client server application (using proxy pattern and RMI).

C. Non Functional Requirements (Based on design/code) (6 marks)

In your solution you must use and document: two *creational*, two *structural*, and two *behavioural* patterns from the Gamma et al. pattern catalogue as covered in lectures 5-10.

For **each** of the three categories (creational, structural and behavioural) you must choose at least one pattern from the following list (these are some of the more complex and/or significant patterns that can be readily applied in a good solution to the assignment). i.e. total of three patterns from the list (one from each category) and three patterns of your own choosing (one from each category), for a total of **SIX** patterns.

Creational
Abstract Factory
Prototype

Structural
Composite
Decorator

Behavioural
Visitor
Observer
Chain of Responsibility
Command
Bridge
Persistent Data Structure

For each of the 6 chosen patterns you should provide a class diagram showing the specific usage in your assignment using the exact naming (class, method names etc.) of your source code so we can cross reference your diagram and the code when marking. Include only classes that directly relate the specific pattern usage. Also for each pattern you should provide a short description of why and how the patterns were used and what advantages it provided in this specific use case. These diagrams and descriptions are to be submitted along with your final code at the end of week 12. (Individual contributions must be highlighted)

Code Quality 6 Marks

- Quality of code based on how effectively you implemented the GoF design patterns.
- Emphasis on high cohesion, low coupling, indirection and good extensibility.
- Final code to meet the code quality guidelines (see appendix A of assignment 1)
- Use of design by contract (class invariants, pre and post-conditions for methods)

[OPTIONAL] Bonus Features **(Up to 3 marks maximum total)**

1. Alternative Universes **(2 marks)**

Allow the user to rewind the current game back to any point in the past and start playing again from that point on (as if all the later turn never happened). Record both the original game and all "alternative" games (i.e. record all fork points and branches). The game history will no longer be a sequence of turns, rather it will be a "**TREE structure**" (forks and branches) of turns.

You will need some form of GUI representing the tree that allows a user to select any point in the tree from which to fork an alternative game. To get full marks you must use an appropriate design pattern that makes the implementation of the turn history very **space efficient**.

2. Multiple Game Rules **(2 marks)**

Allow the user to choose between at least two different sets of **game rules** (e.g. healing by proximity vs healing by energy, power X allows double distance move vs power X allows two moves).

The rules need to apply to more than turns; e.g. capture conditions, winning conditions etc. The logic of the rules must be significantly different between rule sets such that they require **separate class hierarchies** to implement each rule set. It goes without saying that you should design your program to **reuse** as much code as possible and **isolate** the reused code from the game rules code in order to reduce the effect of change.

Appendix A: Code Quality Guidelines

Avoid:

- Redundant/obvious comments that can be inferred from the source code
- Commented out code
- Dead functions
- Large classes and methods
- Data only classes
- Incorrect class hierarchy
- Duplicated code in classes and methods
- Feature envy (see lecture on refactoring)
- Inappropriate intimacy (see lecture on refactoring)
- Vertical separation (see lecture on refactoring)
- Switch statements (use polymorphism wherever possible)
- Long parameter list in methods and interfaces (use encapsulation)

Use / Allow

- Readable and maintainable code
- Meaningful and consistent naming convention
- Low Coupling between classes
- High Cohesion in classes
- Consistent indentation and spacing
- Comments that add value above the source code e.g. explaining algorithms, dependencies or reasons for design choices
- Javadoc for core interfaces/classes, especially where there is a boundary between implementation from different team members